# #Java 命令执行jar 传参

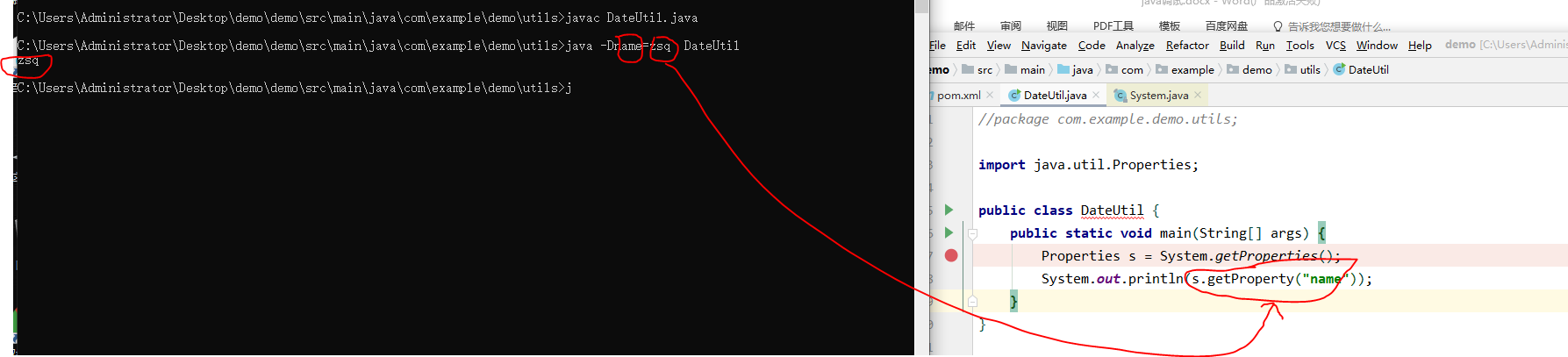
1、传递系统参数

-D<name>=<value> -D作为前缀，设置系统属性。

等价于：setProperty**(**String key, String value**)**，其目的是为了在代码运行前配置一些系统属性。

常见用法：

(1)java -D<name>=<value> javaClass

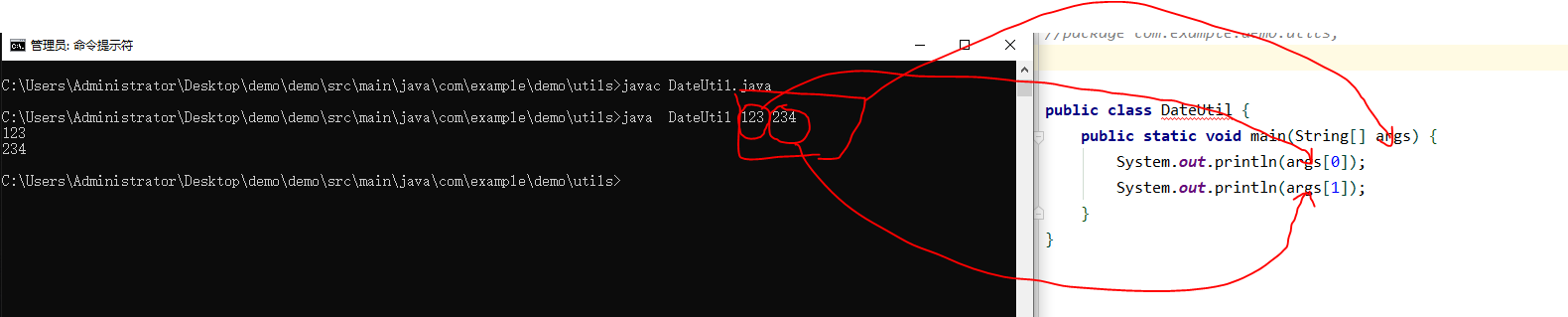


(2)java –Djava.ext.dirs=lib myClass

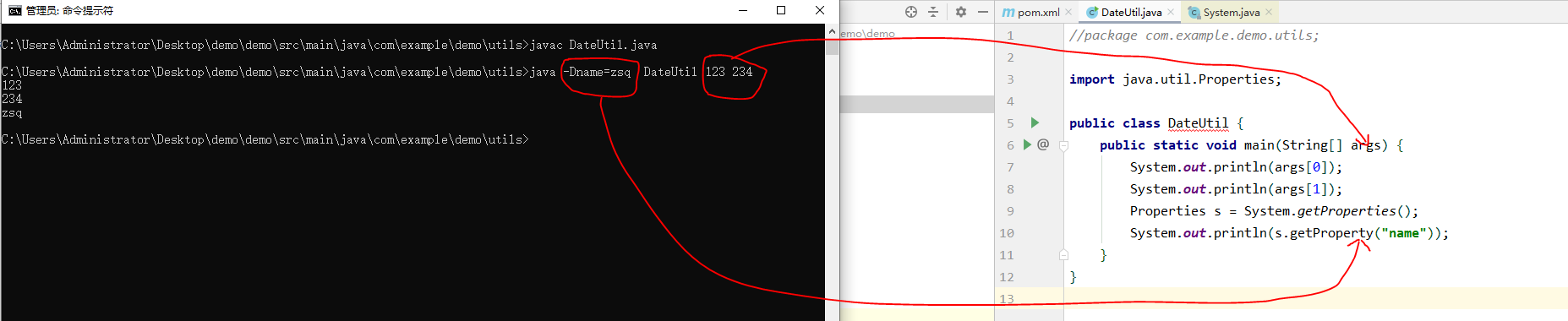
该用法可以简化加入多个jar包，java.ext.dirs是系统默认的系统属性名，表示外部依赖拓展包的目录。

2、传递主函数运行中的参数(args[])

该种传参方式是向主程序传入变量，常用于测试。



两种方式结合一起传参：



注意：系统参数在类名的前面，主函数中的参数在类名的后面。

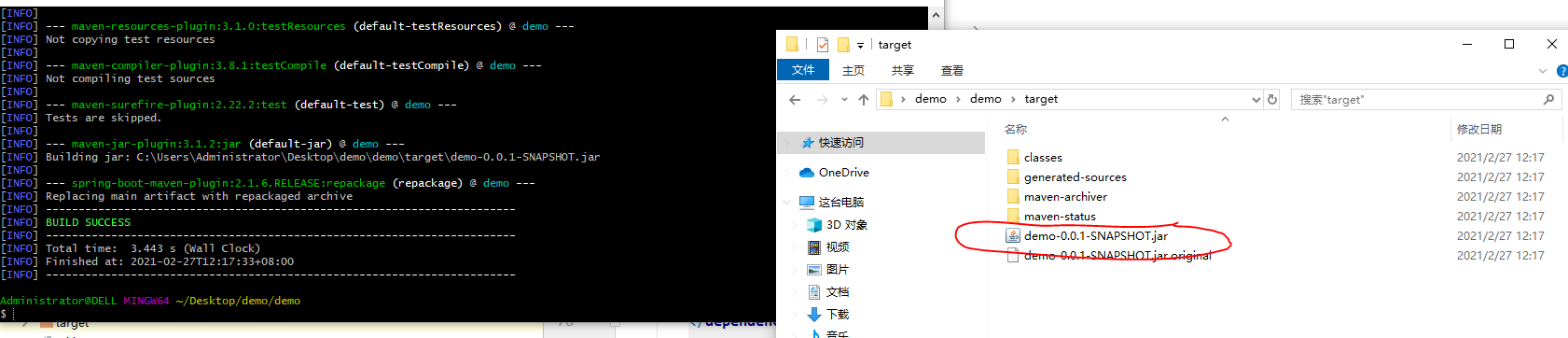
# #SpringBoot项目运行jar传参

1、引入依赖

***<*plugins*>  
 <*plugin*>  
 <*groupId*>***org.springframework.boot***</*groupId*>  
 <*artifactId*>***spring-boot-maven-plugin***</*artifactId*>  
 </*plugin*>  
</*plugins*>***

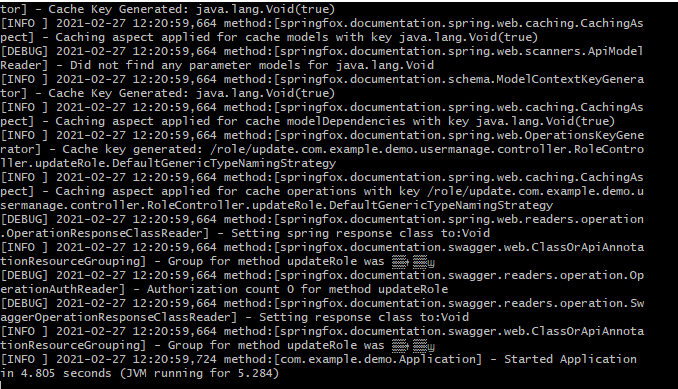
2、打包

mvn clean package -Dmaven.test.skip=true -T 2C



3、运行

java -jar demo-0.0.1-SNAPSHOT.jar



4、传参

java -jar demo-0.0.1-SNAPSHOT.jar --spring.profiles.active=dev

在命令行运行时，连续的两个减号--就是对application.properties中的属性值进行赋值的标识。故上述命令等价于在application.properties中添加属性spring.profiles.active=dev。

为了安全，Springboot提供了屏蔽命令行访问属性的设置，只需要设置：

SpringApplication.setAddCommandLineProperties(false)。

# #引入的其他jar包，命令行运行java程序

用java命令行直接执行程序，如果这个程序需要引用外部jar包；就不能单纯用java xx来执行。

解决方案：

1、javac/java –cp

Windows:

编译：javac -cp D:\yy\yy.jar;D\xx\xx.jar test.java

执行：java -cp D:\yy\yy.jar;D\xx\xx.jar test

Linux:

编译：javac -cp /yy/yy.jar:/xx/xx.jar test.java

执行：java -cp /yy/yy.jar:/xx/xx.jar test

原理：

java 是执行编译后的class的.

-cp 就是引入外部 zip包或者jar包，就相当于在环境变量配置的classpath后的参数。然后到指定的目录找指定运行时需要的class。

缺点：当依赖的包很多时，要一个个填写就很麻烦。接下第二种方案就可以解决这个问题。

2、javac/java -Djava.ext.dirs=

Windows:

编译：javac -Djava.ext.dirs=D:\usrlib;D:\lib test.java

执行：java -Djava.ext.dirs=D:\usrlib;D:\lib test

Linux:

编译：javac -Djava.ext.dirs=/usrlib:/lib test.java

执行：java -Djava.ext.dirs=/usrlib:/lib test

3、-cp 和 -Djava.ext.dirs区别

-cp 和 -classpath 一样，是指定java类运行所依赖其他类的路径，通常是类库，jar包之类，需要全路径到jar包，window上使用分号“;”  分隔，linux上是使用冒号“:”分隔。不支持通配符，也不支持文件夹的方式，需要列出所有jar包，用点“.”代表当前路径，这个如果要引用好多jar包，要一个个加入到路径里，写起来就很麻烦。

-Djava.ext.dirs是通过设置系统属性的方式也加载jar包的，级别就有点高，和-classpath的区别在于-Djava.ext.dirs会覆盖Java本身的ext设置，java.ext.dirs指定的目录由ExtClassLoader加载器加载，如果自定义的程序没有指定该系统属性（-Djava.ext.dirs=sss/lib）那么该加载器默认加载$JAVA\_HOME/lib/ext目录下的所有jar文件。

但如果手动指定系统属性且忘了把$JAVA\_HOME/lib/ext路径给加上，那么ExtClassLoader不会去加载$JAVA\_HOME/lib/ext下面的jar文件，意味着将失去一些功能。

例如java自带的加解密算法实现。具体说是jre的ext目录：D:\java\jdk1.8.0\_40\jre\lib\ext。一般情况下普通程序运行并无差异，因为可以看到ext目录下也就几个jar，但是如果使用java.security相关类时，也就是使用加密解决相关的算法，就会发现-Djava.ext.dirs会导致library找不到相关类而报错。如报错：

NoSuchAlgorithmException: Cannot find any provider supporting RSA

当然也有解决方法：

1、将ext下相关jar包复制到新的ext director。

2、在-D.java.ext.dirs中配置多个目录。可以使用冒号分隔（windows下使用分号）。比如：-Djava.ext.dirs=directoryA:directoryB

# #Java项目中的classpath

开发时期的项目里，文件打包目录如下：

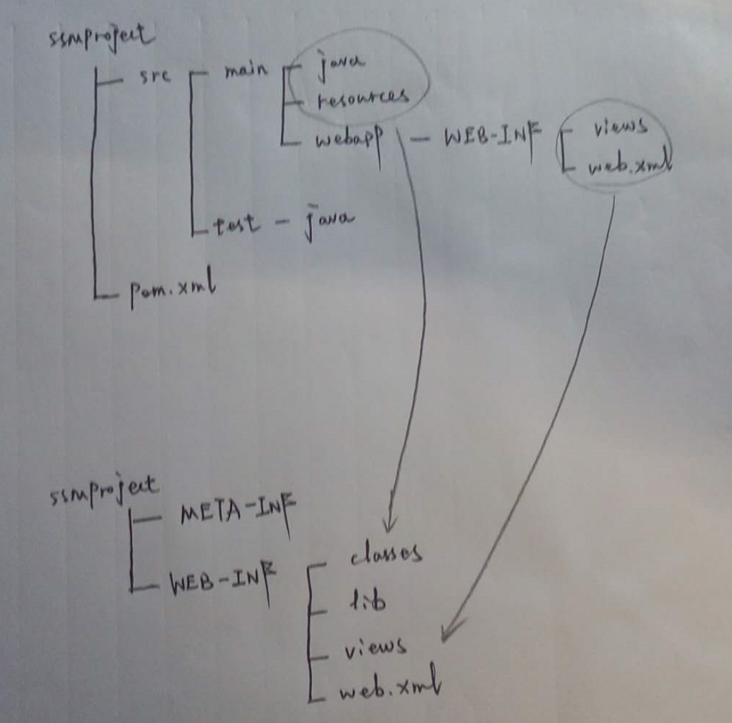
（1）src/main/下面的java和resources文件夹都被(编译)打包到了生产包的WEB-INF/classes/目录下；

（2）而原来WEB-INF下面的views和web.xml则仍然还是在WEB-INF下面。

（3）同时由maven引入的依赖都被放入到了WEB-INF/lib/下面。

（4）最后，编译后的class文件和资源文件都放在了classes目录下。

在编译打包后的项目中，根目录是META-INF和WEB-INF 。这个时候，可以看到classes这个文件夹，它就是要找的classpath。



# #SpringBoot分离部署JAR瘦身优化

使用到的打包插件：

spring-boot-maven-plugin

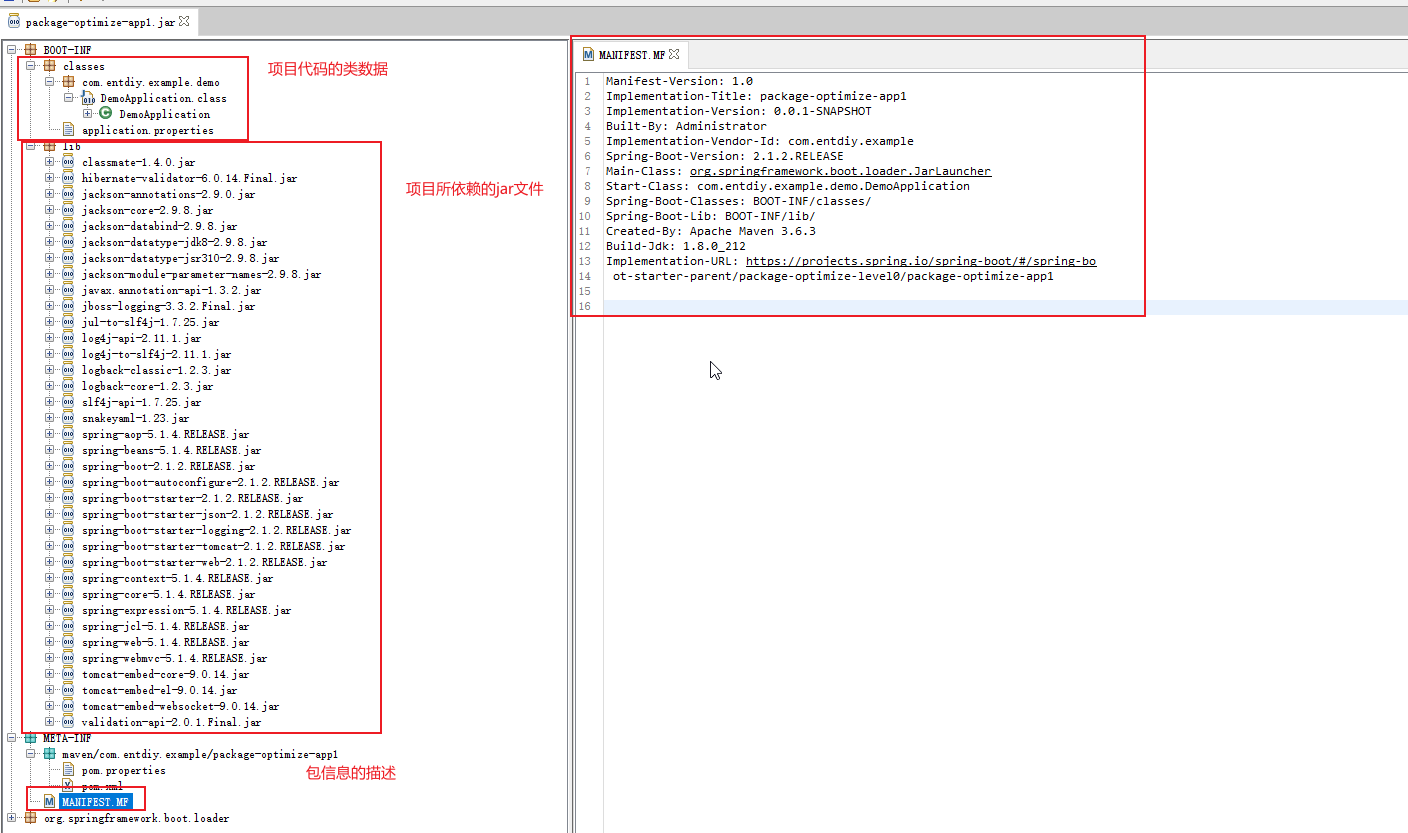
maven-dependency-plugin

maven-jar-plugin

## Level 0:常规的jar构建

主要是利用spring-boot-maven-plugin插件，会把项目源码文件，项目依赖的其他jar，以及资源文件全部打包在一个jar中，大小为15.9MB。下图是通过反编译jar查看的jar包里面的所有文件。

* 实际情况单一构建根据项目依赖组件量输出jar一般在几十MB到一两百MB甚至更大。
* 假如有十来个微服务需要部署，那就意味着需要传输一两个GB的文件，耗时可想而知。就算是单一更新个别微服务也需要传输一两百MB。



## Level 1:常见的依赖jar分离构建方式

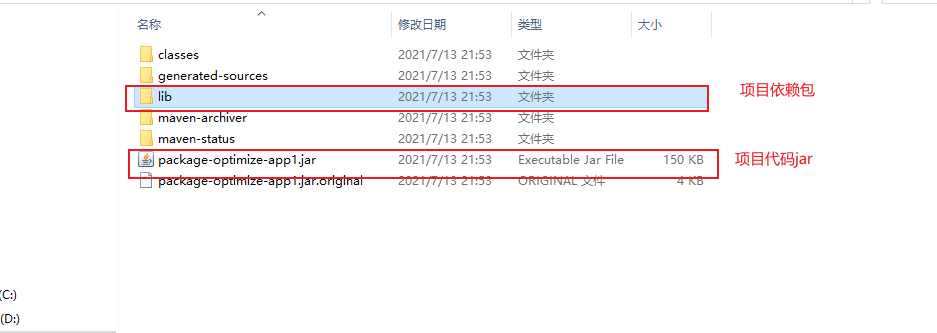
主要利用spring-boot-maven-plugin、maven-dependency-plugin插件;主要解决的问题是：降低单个微服务jar文件的大小，以便部署过程中秒传文件。

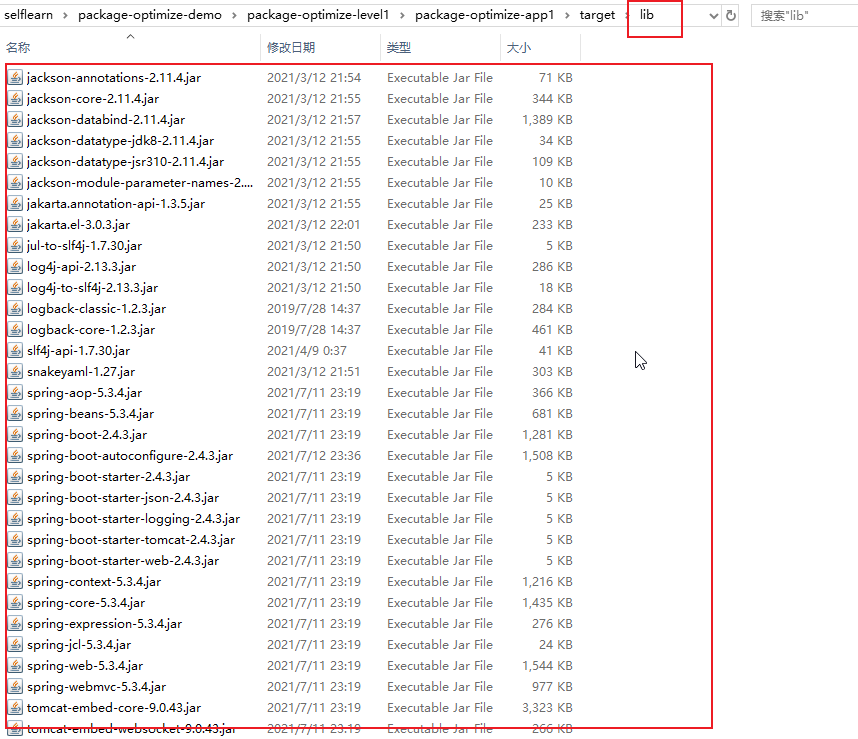
主要配置如下：

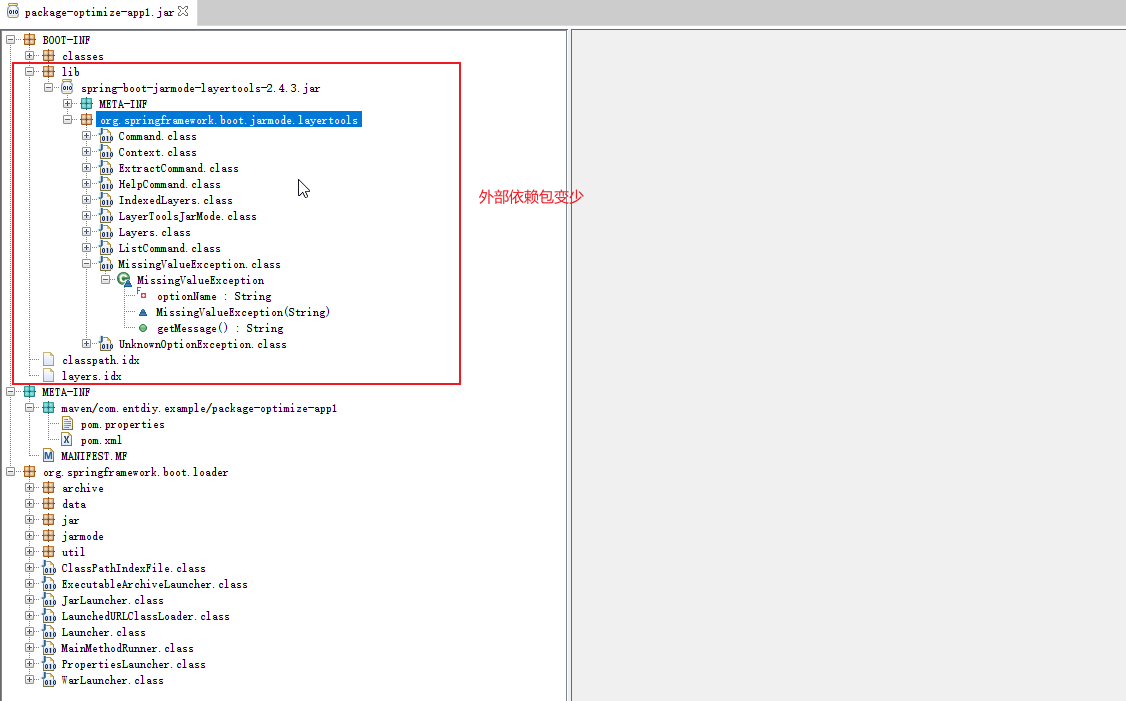
<build>  
    <finalName>${project.artifactId}</finalName>  
    *<!--  
    特别注意：  
    项目仅仅是为了演示配置方便，直接在parent的build部分做了插件配置和运行定义。  
    但是实际项目中需要把这些定义只放到spring boot模块项目（可优化使用pluginManagement形式），避免干扰其他util、common等模块项目  
    -->*  
    <plugins>  
        *<!-- 拷贝项目所有依赖jar文件到构建lib目录下 -->*  
        <plugin>  
            <groupId>org.apache.maven.plugins</groupId>  
            <artifactId>maven-dependency-plugin</artifactId>  
            <executions>  
                <execution>  
                    <id>copy-dependencies</id>  
                    <phase>package</phase>  
                    <goals>  
                        <goal>copy-dependencies</goal>  
                    </goals>  
                    <configuration>  
                        <outputDirectory>${project.build.directory}/lib</outputDirectory>  
                        <excludeTransitive>false</excludeTransitive>  
                        <stripVersion>false</stripVersion>  
                        <silent>true</silent>  
                    </configuration>  
                </execution>  
            </executions>  
        </plugin>  
        *<!-- Spring Boot模块jar构建 -->*  
        <plugin>  
            <groupId>org.springframework.boot</groupId>  
            <artifactId>spring-boot-maven-plugin</artifactId>  
            <configuration>  
                <includes>  
                    *<!-- 不存在的include引用，相当于排除所有maven依赖jar，没有任何三方jar文件打入输出jar -->*  
                    <include>  
                        <groupId>null</groupId>  
                        <artifactId>null</artifactId>  
                    </include>  
                </includes>  
                <layout>ZIP</layout>  
            </configuration>  
            <executions>  
                <execution>  
                    <goals>  
                        <goal>repackage</goal>  
                    </goals>  
                </execution>  
            </executions>  
        </plugin>  
    </plugins>  
</build>

打包以后target目录、lib目录以及jar截图如下：

* 从图中可以看出单一构建根据项目依赖组件量输出jar一般仅有一两百KB，基本可以做到秒传。
* 假如有十来个微服务，每个服务一个jar和一个lib目录文件，首次部署也差不多需要传输一两个GB文件。
* java -jar -Djava.ext.dirs=lib package-optimize-app1/target/package-optimize-app1.jar 启动命令
* Level0与Level1最大区别就是jar包中lib包变简洁了，而且依赖从jar内部转移到外部.







## Level 2:合并所有模块依赖jar到同一lib目录

解决问题：

* 合并所有模块依赖jar到同一个lib目录，一般由于各模块项目依赖jar重叠程度很高，合并所有服务部署文件总计大小基本也就两三百MB
* 但是如果采用-Djava.ext.dirs=lib加载所有jar到每个JVM，一来每个JVM都完整加载了所有jar耗费资源，二来各微服务组件版本不同会出现版本冲突问题

主要配置如下：

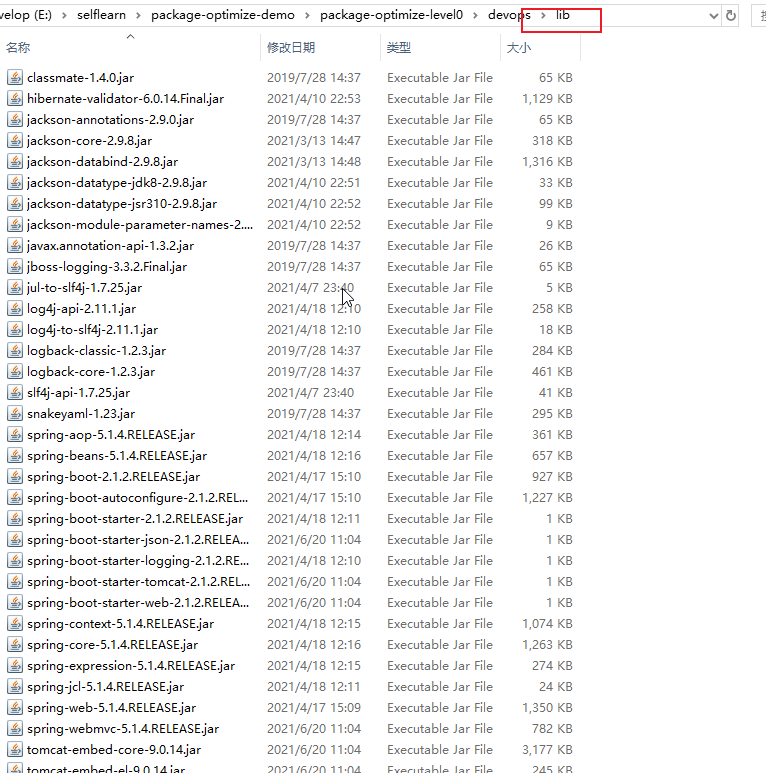
<build>  
    <finalName>${project.artifactId}</finalName>  
    *<!--  
    特别注意：  
    项目仅仅是为了演示配置方便，直接在parent的build部分做了插件配置和运行定义。  
    但是实际项目中需要把这些定义只放到spring boot模块项目（可优化使用pluginManagement形式），避免干扰其他util、common等模块项目  
    -->*  
    <plugins>  
        *<!-- 基于maven-jar-plugin插件实现把依赖jar定义写入输出jar的META-INFO/MANIFEST文件 -->*  
        <plugin>  
            <groupId>org.apache.maven.plugins</groupId>  
            <artifactId>maven-jar-plugin</artifactId>  
            <configuration>  
                <archive>  
                    <manifest>  
                        <addClasspath>true</addClasspath>  
                        <classpathPrefix>lib/</classpathPrefix>  
                        <useUniqueVersions>false</useUniqueVersions>  
                    </manifest>  
                </archive>  
            </configuration>  
        </plugin>  
        *<!-- 拷贝项目所有依赖jar文件到构建lib目录下 -->*  
        <plugin>  
            <groupId>org.apache.maven.plugins</groupId>  
            <artifactId>maven-dependency-plugin</artifactId>  
            <executions>  
                <execution>  
                    <id>copy-dependencies</id>  
                    <phase>package</phase>  
                    <goals>  
                        <goal>copy-dependencies</goal>  
                    </goals>  
                    <configuration>  
                        *<!--  
                        各子模块按照实际层级定义各模块对应的属性值，检查所有微服务模块依赖jar文件合并复制到同一个目录  
                        详见各子模块中 boot-jar-output 属性定义  
                        -->*  
                        <outputDirectory>${boot-jar-output}/lib</outputDirectory>  
                        <excludeTransitive>false</excludeTransitive>  
                        <stripVersion>false</stripVersion>  
                        <silent>false</silent>  
                    </configuration>  
                </execution>  
            </executions>  
        </plugin>  
        *<!-- Spring Boot模块jar构建 -->*  
        <plugin>  
            <groupId>org.springframework.boot</groupId>  
            <artifactId>spring-boot-maven-plugin</artifactId>  
            <configuration>  
                <includes>  
                    *<!-- 不存在的include引用，相当于排除所有maven依赖jar，没有任何三方jar文件打入输出jar -->*  
                    <include>  
                        <groupId>null</groupId>  
                        <artifactId>null</artifactId>  
                    </include>  
                </includes>  
                <layout>ZIP</layout>  
                *<!--  
                基于maven-jar-plugin输出微服务jar文件进行二次spring boot重新打包文件的输出目录  
                所有微服务构建输出jar文件统一输出到与lib同一个目录，便于共同引用同一个lib目录  
                详见各子模块中boot-jar-output属性定义  
                -->*  
                *<!--  -->*  
                <outputDirectory>${boot-jar-output}</outputDirectory>  
            </configuration>  
            <executions>  
                <execution>  
                    <goals>  
                        <goal>repackage</goal>  
                    </goals>  
                </execution>  
            </executions>  
        </plugin>  
    </plugins>  
</build>

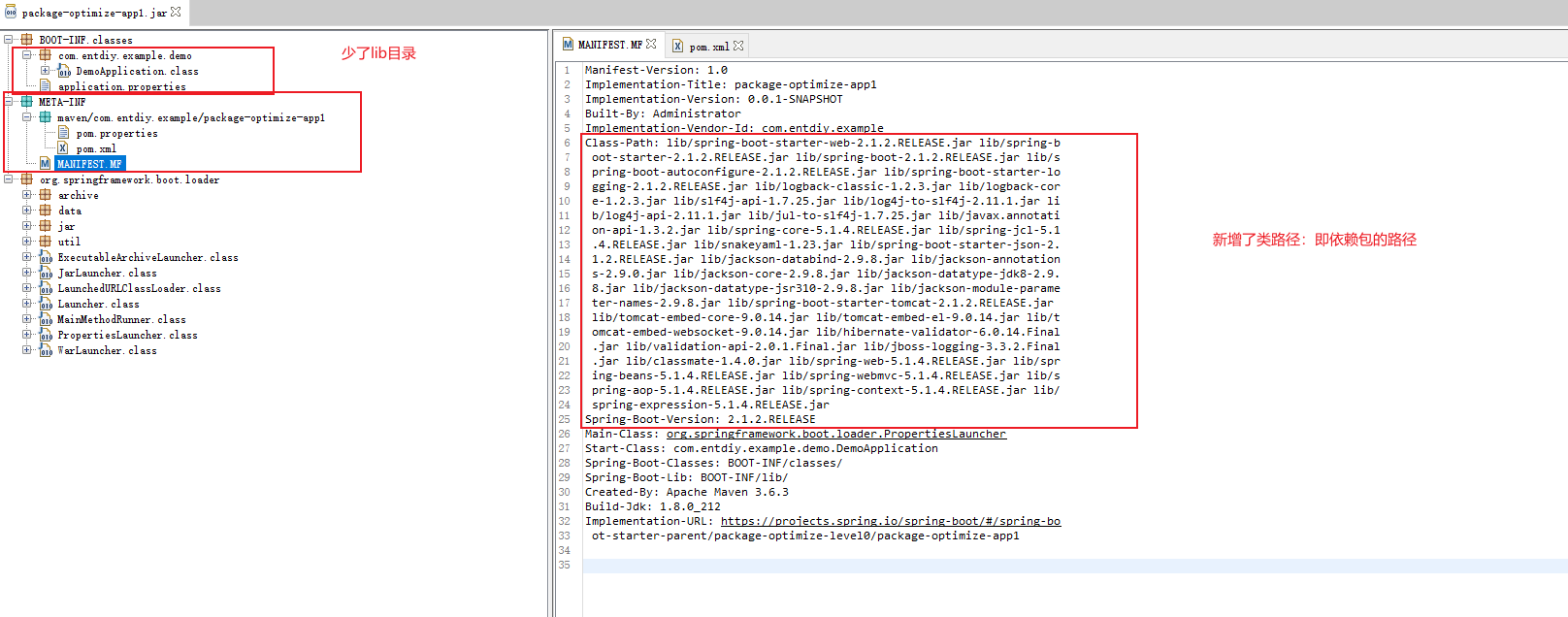
打包以后的jar包、lib目录截图如下：

可得：

* 三个微服务打成jar以后，都很小；而且三个微服务的依赖包全部打包到lib目录下
* 启动过程不再需要 -Djava.ext.dirs=lib 参数定义。
* java -jar devops/package-optimize-app1.jar 启动命令
* 通过定制每个微服务jar文件中的META-INFO/MANIFEST文件中的Class-Path明确指明依赖版本组件类，解决各微服务不同组件版本冲突问题。
* Level 2 与 Level 1、Level 0最大的区别是：META-INFO/MANIFEST文件新增了class-path属性；此外BOOT-INFO目录下少了lib目录。







详细描述参考：

https://mp.weixin.qq.com/s/ciR6Yk0DmYvIe9HARM7IDA

https://gitee.com/xautlx/package-optimize-demo